Deep Learning Cheatsheet

Author: Gonçalo Nunes
Revised by Dinis Felgueiras

January 19, 2025

1 Optimization

A function f is **convex**, if, for any λ ∈[0, 1], and any x, x': f(λx + (1 - λ)x') ≤ f(λx) + (1 - λ)f(x'). Intuitively, if we draw a line between two points in f, then all linear combination of points between them are below the line or on the line. A function f is **strictly convex**, if f(λx + (1 - λ)x') ≤ f(λx) + (1 - λ)f(x') (any linear combination of points is exactly below the line).

1.1 Gradient Descent

Take small steps in the negative gradient direction until a stopping criterion is met: $x^{t+1} \leftarrow x^t - \eta_t \nabla f(x^t)$

1.1.1 Stochastic Gradient Descent

Do the updates by computing the gradient one example at a time (pick example randomly)

1.1.2 Batch Gradient Descent

Pick a subset of examples, instead of just 1. If the size of the subset is 1, we get SGD.

1.2 Optimizers

AdaGrad:

- **Advantages:** Robust to the choice of α and to different parameter scaling.
- **Drawbacks:** Step size vanishes because $G_{j,t} \geq G_{j,t-1}$.

RMSProp:

- **Advantages:** Robust to the choice of α (typically 0.01 or 0.001); robust to different parameter scaling.

Adam:

- **Advantages:** Computationally efficient, low memory usage, suitable for large datasets and many parameters.
- **Drawbacks:** Possible convergence issues and noisy gradient estimates.

2 Linear Models

Least Square Methods: $min_{w,b} \sum_{n=1}^{N} (\hat{y} - (Wx_n + b)^2)$
Least squares is equivalent to Maximum Likelihood Estimation
Squared Loss: $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$
Absolute error: $L(y, \hat{y}) = |y - \hat{y}|$
Data is **linearly separable** if there exists (w, b) such that classification is perfect

2.1 Linear Regression

Model: $y = W^T \phi(x) + b$
Closed Form: $\hat{W} = (X^T X)^{-1} X^T y$
Generalize with a design matrix $\Phi(x)$ by adding a column of 1s at the start and w_0 = b to the weight matrix.

2.1.1 Regularization

Ridge regression (closed form): $\hat{W} = (X^TX + \lambda I)^{-1}X^Ty$

The use of Regularization, like L_2, is a way to mitigate overfitting.

Dropout Regularization: During training, remove some hidden units, chosen at random (each hidden unit output is set to 0 with probability p). This prevents hidden units to co-adapt to other units, forcing them to be more generally useful. At test time, keep all units, with the outputs multiplied by 1 - p. *Hidden layer* becomes $h^l(x) = g(z^l(x)) \odot m^l$. *Gradient with respect to* z^{l-1} becomes $\nabla_{z^{l-1}}L(f(x;\theta), y) = \nabla_{h^{l-1}}L(f(x;\theta), y) \odot g'(z^{l-1}) \odot m^{l-1}$

Inverted Dropout: Like regular dropout regularization, at training time the output of the units that were not dropped is divided by 1 - p and requires no change at test time

2.2 Linear (Binary) Classifier

$$y = sign(W^T\phi(x) + b) = \begin{cases} +1 & W^T\phi(x) + b \geq 0 \\ -1 & W^T\phi(x) + b < 0 \end{cases} \tag{1}$$

Decision boundary is a hyperplane, with regards to $\phi(x)$ and not x: $W^T\phi(x) + b = 0$

2.3 Perceptron Algorithm

1. Apply current model to x_n, get the corresponding prediction
2. If prediction is correct, do nothing
3. If it is wrong, correct w by adding/subtracting feature vector $\phi(x_i)$

Classify as a linear binary classifier with a sign activation. Update weights with $w^{(k+1)} = w^{(k)} + y_n\phi(x_n)$. Perceptrons can only learn linearly separable sets.

2.3.1 Perceptron Mistake Bound

The training data is linearly separable with margin $\gamma > 0$ iff there is a weight vector u, with $||u|| = 1$, such that:

$$y_n u^T\phi(x_n) \geq \gamma, \forall n$$

Radius of the data: R = $max_n||\phi(x_n)||$

The perceptron algorithm is guaranteed to find a separating hyperplane after at most $\frac{R^2}{\gamma^2}$ mistakes.

2.4 Logistic Regression

A linear model gives a score for each class $y : W^T\Phi(x)$ from which we can compute a conditional (posterior) probability: softmax($W^T\Phi(x)$) = P(y—x) = $\frac{e^{W_y^T\Phi(x)}}{\sum_{y'\in Y} e^{W_{y'}^T\Phi(x)}}$

Equivalent to a linear classifier, if we chose the *most probable class*. For binary classification, softmax becomes the sigmoid function. In a multi-nomial logistic regression, we learn the weights by maximizing the conditional log-likelihood. This a convex optimization problem, but there is no closed form solution.

3 Neural Networks

(neuron) Pre-Activation: $z(x) = W^Tx + b = \sum_{i=1}^{D} w_ix_i + b$
(neuron) Activation: $h(x) = h(z(x)) = h(W^Tx + b)$

3.1 Activation functions

Linear Activation: $h(z) = z$; Useful to project the input to a lower dimension. No squashing of the input.

Sigmoid Activation: $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$; Squashes the neuron pre-activation between 0 and 1. The output can be interpreted as a probability. Positive, bounded, strictly increasing. Logistic regression corresponds to a network with a single sigmoid unit. Combining layers of sigmoid units increases expressivenes. Easy to saturate and, when that happens, the corresponding gradients are only residual, making learning slower

Hyperbolic Tangent Activation: $g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Squashes the neuron pre-activation between -1 and 1. Bounded, strictly increasing. Combining layers of tanh units increases expressivenes.

Rectified Linear Unit: relu(z) = max(0,z). Non-negative, increasing, but not upper bounded. Not differentiable at 0. Leads to neurons with sparse activities. Less prone to vanishing gradients. Saturate only for negative inputs, and have constant gradient for positive inputs, often exhibiting fast learning

3.2 Feed-Forward Network Backpropagation Formulas

K_l refers to the number of hidden units, l refers to the number of the layer.
Hidden layer pre-activation: $z^l(x) = W^l h^{l-1} + b^l$; with $W^l \in \mathbb{R}^{K_l x K_{l-1}}$ and $h^{l-1} \in \mathbb{R}^K$
Hidden layer activation: $h^l(x) = g(z^l(x))$
Output layer activation: $f(x) = o(z^{l+1}(x)) = o(W^{L+1}h^L + b^{L+1})$

3.2.1 Gradients

$\nabla_{W^l} L(f(x;\theta), y) = \nabla_{z^l} L(f(x;\theta), y) h^{(l-1)^T}$
$\nabla_{b^l} L(f(x;\theta), y) = \nabla_{z^l} L(f(x;\theta), y)$
$\nabla_{h^{l-1}} L(f(x;\theta), y) = W^{(l)^T} \nabla_{z^l} L(f(x;\theta), y)$
$\nabla_{z^{l-1}} L(f(x;\theta), y) = \nabla_{h^{l-1}} L(f(x;\theta), y) \odot g'(z^{l-1})$
Lastly,
$W_{new} = W_{old} - \eta \nabla_{W_L}$

4 Convolutional Neural Networks

In CNN's we do the multiplication element-wise.
CNNs are very useful because they preserve spatial structure and can recognize the same features, even if they "change place".

Why choose ReLU's in CNN's? ReLUs are significantly simpler and have a much simpler derivative than the sigmoid, leading to faster computation times. Also, sigmoids are easy to saturate and, when that happens, the corresponding gradients are only residual, making learning slower. ReLUs saturate only for negative inputs, and have constant gradient for positive inputs, often exhibiting faster learning.

4.1 Convolution Layer

A key part of CNNs is their convolution layers, where they convolve (slide over, computing dot products) one or more filters/kernels over their input and pass that to the next layer. The term convolution of a signal x and a filter w is defined as h[t] = $(h * w)$[t] = $\sum_{a=-\infty}^{\infty} x[t-a]w[a]$

The terms channels, filters and kernels are used somewhat interchangeably. The number of channels is the number of filters used in a layer.

This type of layer allows us to **achieve a sparser/more local connectivity**, but sharing parameters, as opposed to what we'd get in a fully connected network. This **parameter sharing** leads to a smaller complexity (less parameters to learn), **translation/shift equivariance**, which means the order in which we apply translations/shifts to an image doesn't matter (we get the same end result) and allows dealing with arbitrary large, variable-length, inputs (don't shift the filters, shift the input).

In certain CNN's, 1×1 convolutional layer is used (GoogleNet). This maps each input "pixel" (with all its channels) to one output pixel in a nonlinear way. It can be thought of as applying a fully connected layer to each input "pixel". This can be used, for example, to reduce or enlarge the number of channels in an image in a "pixelwise" manner, performing a nonlinear transformation on each of them.

4.2 Pooling Layer

Pooling layers provide **invariance**, this means that if we, for example, scale an image, our CNN will still predict the same thing. They also **make the representations smaller**.
The most common type of pooling is **max-pooling**, where the result of applying the pooling filter is the maximum value it encounters. Max pooling offers **scaling, shifting and rotation invariance**.

4.3 Variable Length Input and Padding

Most computation in CNNs can be done in parallel, but, unlike images, which have fixed size, sentences have different lengths, which makes batching a bit trickier! If we just add padding mindlessly, we might get a lot of waste if we have a very large sentence in a batch, together with a very small one (the small one will need lots of padding)
Solution: minimize waste by sorting by sentence length before forming mini-batches, then padding (masking is needed to ensure the padding is not affecting the results)

TODO - padding lecture 7

4.4 Complexity and Output Size

Given an N × N × D image, F × F × D filters, K channels, and stride S, the resulting output will be of size M × M × K , where $M = \frac{N-F}{S} + 1$

W - image dimension, S - stride, K kernel size, P padding
$\lceil \frac{W-K+2P}{S} \rceil + 1$
`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`
Examples:

- N = 32, D = 3, F = 5, K = 6, S = 1 results in an 28 × 28 × 6 output
- N = 32, D = 3, F = 5, K = 6, S = 3 results in an 10 × 10 × 6 output

Output Width $= \frac{InputWidth - KernelWidth + 2 \times PaddingWidth}{Stride} + 1$

Number of Units Within a Layer $= OutputWidth \times OutputHeight \times NumberOfFilters$

Number of trainable weights $= NumberOfFilters \times ((KernelWidth \times KernelHeight \times NumberOfChannels) + Bias)$

Number of Trainable Weights (formula 2): $(KernelSize)^2 \times NumberOfInputChannels \times NumberOfOutputChannels$

Number of Biases: Number Of Output Channels or Filters

Number of parameters if we have a FF instead of CNN $= NumberOfunits \times ((InputWidth \times InputHeight \times NumberOfChannels) + Bias)$

```
nn.Sequential(
    nn.Conv2d(1, 5, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(5, 10, kernel_size=5, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(10, 20, kernel_size=2, stride=2, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=5, stride=2),
    nn.Flatten(),
    nn.Linear(2800, 6))
```

Fill in the following table with the adequate values.

Layer	Output size	N. weights	N. biases
Input	$192 \times 256 \times 1$	0	0
1st conv. layer	$192 \times 256 \times 5$	45	5
1st pooling layer	$96 \times 128 \times 5$	0	0
2nd conv. layer	$92 \times 124 \times 10$	1250	10
2nd pooling layer	$46 \times 62 \times 10$	0	0
3rd conv. layer	$23 \times 31 \times 20$	800	20
3rd pooling layer	$10 \times 14 \times 20$	0	0
Output layer	6×1	16,800	6

Figure 1: Solved Exercise

5 Recurrent Neural Networks

$$h_t = g(Vx_t + Uh_{t-1} + c)$$
$$\hat{y}_t = Wh_t + b$$

If we unroll a RNN's computation graph we get a directed acyclic graph, so we can **train it by backpropagating gradients**. Parameters are tied/shared accross "time" and derivatives are aggregated across time steps, this is called **backpropagation through time (BPTT)**. In BPTT, the gradient behaviour changes drastically according to the largest eigenvalue of weight matrix **U**:

- largest eigenvalue of U exactly 1: gradient propagation is stable
- largest eigenvalue of U $<$ 1: gradient vanishes (exponential decay)
- largest eigenvalue of U $>$ 1: gradient explodes (exponential growth)

Exploding Gradients: can be dealt with by gradient clipping (truncating the gradient if it exceeds some magnitude)

Vanishing Gradients: are more frequent and harder to deal with, in practice: long-range dependencies are difficult to learn. Possible solutions: normalization to stabilize gradients, clever initialization, LSTMs and Gated Recurrent Units. Intuition for LSTM/GRU: instead of multiplying across time (which leads to exponential growth), we want the error to be approximately constant

Some ways to solve this:

- ReLU good for this because it's derivative is constant, 1.
- skipping connections

Training RNN's using gradient descent can suffer from the vanishing gradients problem, where gradients become very small and fail to update weights effectively for long sequences. This makes it difficult to learn long-range dependencies.

Parameter tying: $\frac{\partial L}{\partial U} = \sum_{t=1}^{numTimeSteps} \frac{\partial h_t}{\partial U} \frac{\partial L}{\partial h_t}$

5.1 Sequence Generation

Generating each word depends on all the previous words, but this doesn't scale well, we would need too many parameters. One solution is to use limited memory with an order-m Markov model (n-grams). Alternative: consider all the history, but compress it into a vector (this is what RNNs do). Sequence generataion RNNs are typically trained with **maximum likelihood estimation** (cross-entropy), so it intuitively measures how "perplexed/surprised" the model is.

We can also have an RNN over characters instead of words. Advantage: can generate any combination of characters, not just words in a closed vocabulary and much smaller set of output symbols; Disadvantage: need to remember deeper back in history

5.1.1 Auto Regressive Models

The auto-regressive model specifies that the output variable depends linearly on its own previous values and on a stochastic term (an imperfectly predictable term). The key ideas are:

1) Feed the previous output as input to the current step: $x_t = embedding of y_{t-1}$

2) Maintain a state vector h_t , which is a function of the previous state vector and the current input: this state compresses all the history. $h_t = g(Vx_t + Uh_{t-1} + c)$

3) Compute next output probability: $P(y_t = i|y_{t-1}, ..., y_0 = softmax_i(Wh_t + b)$

5.1.2 Teacher Forcing and Exposure Bias

In **Teacher Forcing**, we feed the real label y_t of time step t to the next step, during the training. Teacher forcing causes **exposure bias** at run time: the model will have trouble recovering from mistakes early on, since it generates histories that it has never observed before (at testing time we don't have the target sentence to provide as input to the decoder)

5.2 Sequence Tagging

Given an input sequence $<x_1, x_2, ..., x_L>$, assign a tag to each element, producing an output sequence $<y_1, y_2, ..., y_L>$. Differences from sequence generation: the input and output are distinct (no need for auto-regression) and the length of the output is known (same as that of the input).

5.2.1 Bidirectional RNN

Instead of reading a sequence left-to-right to obtain a representation, read in both directions!

5.3 Pooled Classification

Instead of generating an output sequence, predict a single label for the whole sequence, just pool the RNNs states, i.e., map them to a single vector and use a single softmax to output the final label.

The simplest strategy is to **use the last RNN state**. **Disadvantage**: for long sequences, the influence the earliest words have may vanish. Other strategies include average pooling and using a bidirectional RNN and combine both last states of the left-to-right and right-to-left RNNs.

5.4 Gated Recurrent Units

Idea: create some kind of shortcut connections / create adaptive shortcuts controlled by special gates

$$h_t = u_t \odot \hat{h} + (1 - u_t) \odot h_{t-1}$$

Candidate Update: $\hat{h} = g(Vx_t + U(r_t \odot h_{t-1}) + b)$
Reset Gate: $r_t = \sigma(V_r x_t + U_r h_{t-1} + b_r)$
Update Gate: $u_t = \sigma(V_u x_t + U_u h_{t-1} + b_u)$

5.4.1 Long-Short Term Memory

Combats Vanishing Gradients Problem.

Key Ideas: use memory cells c_t; to avoid the multiplicative effect, flow information additively through these cells, control the flow with special input, forget, and output gates.

$$c_t = f_t \odot c_{t-1} + i_t \odot g(Vx_t + Uh_{t-1} + b), h_t = o_t \odot g(c_t)$$

Forget Gate: $f_t = \sigma(V_f x_t + U_f h_{t-1} + b_f)$
Input Gate: $i_t = \sigma(V_i x_t + U_i h_{t-1} + b_i)$
Output Gate: $o_t = \sigma(V_o x_t + U_o h_{t-1} + b_o)$

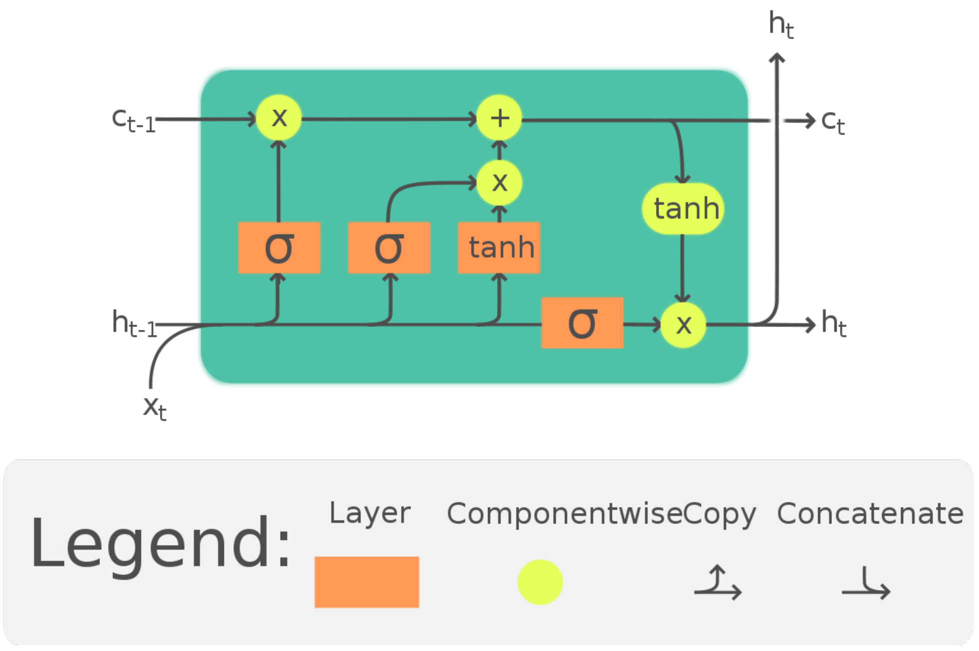


Figure 2: LSTM architecture

BiLSTMs can capture dependencies in both forward and backward directions of the sequence, providing the model with a more comprehensive understanding of the context.

5.5 Como resolver exercícios de RNNs

$$z_t = W_{hh}h_{t-1} + W_{hx}x_t + b_h$$

$$h_t = g(z_t)$$

$$LabelProbability = p_t = softmax(z_t)$$

$$\hat{y}_t = argmax(W_{yh}h_t$$

$$CrossEntropyLoss = l_t = log[p_t]_{y_t} = -e_y \cdot log(p_t)$$

greedy decoding, resulting words from RNN

6 Transformers and Sequence-To-Sequence Models

Sequence-to-sequence models map a source sequence (of arbitrary length) into a target sequence (also of arbitrary length), This differs from sequence tagging, where the two sequences are of the same length.

6.1 Statistical Machine Learning

Key Idea: use Bayes' law to break this down into two components: **Translation Model** and textbfLanguage Model. **Translation Model**: models how words/phrases are translated (learnt from parallel data); **Language Model**: models how to generate fluent English (learn from monolingual data).
Learning a Language Model requires large amounts of monolingual data and can be done with a markov model (n-gram) or neural networks. Learning a Translation Model requires large amounts of parallel data.

6.2 Neural Machine Translation

Basically, Machine Translation with a single neural network. End-to-end training with parallel data (no more complex pipelines!). The underlying architecture is an encoder-decoder (also called a sequence-to-sequence model)

General idea: Encoder RNN encodes source sentence, generating a vector state and Decoder RNN generates target sentence conditioned on vector state.

We want to find the target sentence y that most resembles a likely translation for our input sequence x. However, **enumerating y is intractable**. One possible (approximate) strategy is **Beam Search**: At each decoder step, keep track of the k (k is the beam size) most probable partial translations; discard the rest. Linear runtime as a function of beam size k. For k = 1, we have **greedy search**

6.2.1 Attention

Generation from Matrices with Attention:

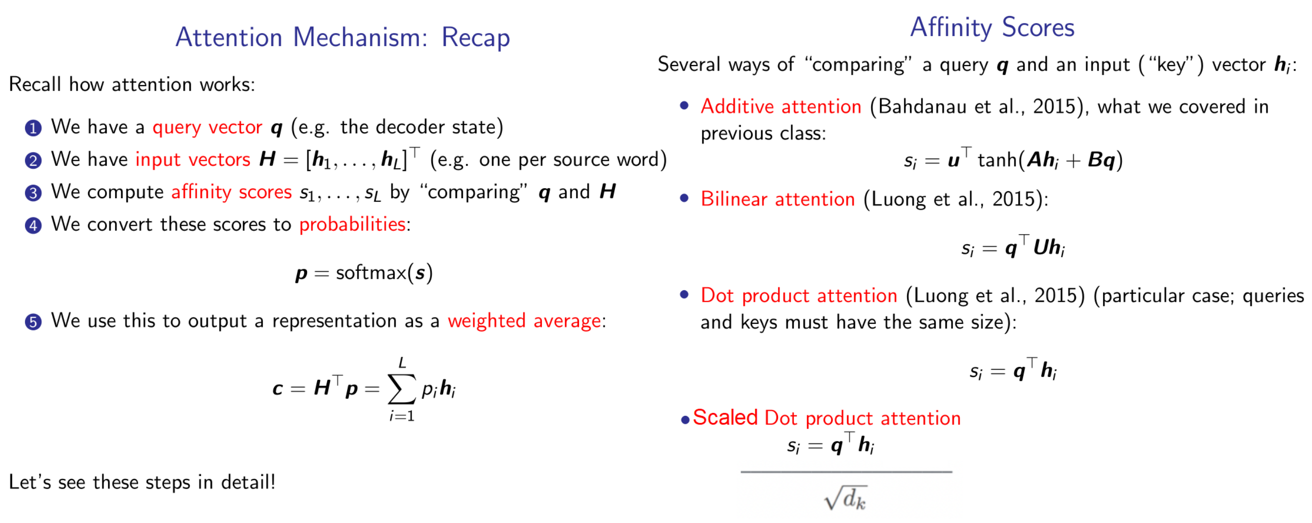
- Generate the output sentence word by word using an RNN
- At each output position t, the RNN receives two inputs: a fixed-size vector embedding of the previous output symbol y_{t-1} and a fixed-size vector encoding a "view" of the input matrix **F** , via a weighted sum of its columns (i.e., words) Fa_t
- The input columns weighting at each time-step (a_t) is called the attention distribution

6.2.2 Attention Mechanism

Let $s_1, s_2, \ldots$ be the states produced by the decoder RNN. When predicting the t-th target word:

- Compute "similarity" with each of the source words $z_{t,i} = v^T g(Wh_i + Us_{t-1} + b)$, for i = 1,...,L, where h_i is the i-th column of F (representation of the i-th source word) and v , W , U, b are model parameters
- From $z_t = (z_{t,1}, ..., z_{t,L})$ compute attention $a_t = \text{softmax}(z_t)$
- Use attention to compute input conditioning state $c_t = Fa_t$
- Update RNN state s_t based on s_{t-1}, y_{t-1}, c_t
- Predict y_t roughly $p(y_t|s_t) = softmax(Ps_t + b)$

It is very useful to allow the decoder to focus on parts of the source; Attention solves the bottleneck problem, by allowing the decoder to look directly at the source; Attention helps with vanishing gradient problem (provides shortcut to faraway states); Attention allows for some interpretability (shows what the decoder was focusing on when producing each output symbol)



Attention Mechanism: Recap

Recall how attention works:

1. We have a query vector $\boldsymbol{q}$ (e.g. the decoder state)
2. We have input vectors $\boldsymbol{H} = [\boldsymbol{h}_1, \dots, \boldsymbol{h}_L]^\top$ (e.g. one per source word)
3. We compute affinity scores $s_1, \dots, s_L$ by "comparing" $\boldsymbol{q}$ and $\boldsymbol{H}$
4. We convert these scores to probabilities:

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{s})$$

5. We use this to output a representation as a weighted average:

$$\boldsymbol{c} = \boldsymbol{H}^\top \boldsymbol{p} = \sum_{i=1}^{L} p_i \boldsymbol{h}_i$$

Let's see these steps in detail!

Affinity Scores

Several ways of "comparing" a query $\boldsymbol{q}$ and an input ("key") vector $\boldsymbol{h}_i$:

- Additive attention (Bahdanau et al., 2015), what we covered in previous class:

$$s_i = \boldsymbol{u}^\top \tanh(\boldsymbol{A}\boldsymbol{h}_i + \boldsymbol{B}\boldsymbol{q})$$

- Bilinear attention (Luong et al., 2015):

$$s_i = \boldsymbol{q}^\top \boldsymbol{U}\boldsymbol{h}_i$$

- Dot product attention (Luong et al., 2015) (particular case; queries and keys must have the same size):

$$s_i = \boldsymbol{q}^\top \boldsymbol{h}_i$$

- Scaled Dot product attention

$$s_i = \frac{\boldsymbol{q}^\top \boldsymbol{h}_i}{\sqrt{d_k}}$$

Figure 3: Attention

6.2.3 Self-Attention

RNNs (even with an attention mechanism) have some drawbacks: sequential mechanism prohibits parallelization and long-range dependencies are tricky, despite gating; **Possible solution**: replace RNN encoder by hierarchical 1-D CNN (encoder becomes parallelizable, but decoder is still sequential and the model is auto-regressive).

(Contextual) Attention recap: Attention allows focusing on an arbitrary position in the source sentence, shortcutting the computation graph.

- We have a query vector q (e.g. the decoder state)
- We have input vectors H = $[h_1, ..., hL]^T$ (e.g. one per source word). Used as keys and values
- We compute affinity scores $s_1, ..., s_L$ by "comparing" q and H
- We convert these scores to probabilities: $p = softmax(s)$
- We use this to output a representation as a weighted average: $c = H^T p = \sum_{i=1}^{L} p_i h_i$ Context Vector

Affinity Scores - Several ways of "comparing" a query q and an input ("key") vector h_i:

- Additive attention: $s_i = u^T tanh(Ah_i + Bq$
- Bilinear attention: $s_i = q^T U h_i$
- Dot product attention (queries and keys must have the same size): $s_i = q^T h_i$

In self-attention, at each position, the encoder attends to the other positions in the encoder itself

Self-Attention Layer for a sequence of length L:

- Query vectors Q = $[q_1, ..., q_L]^T \in \mathbb{R}^{L \times d_Q}$
- Image vector = Image feature matrix^T $*$ attention probabilities
- logits = $A *$ [query vectorimage vector]$^T + b$
- Key vectors K = $[k_1, ..., k_L]^T \in \mathbb{R}^{L \times d_k}$
- Value vectors V = $[v_1, ..., v_L]^T \in \mathbb{R}^{L \times d_v}$
- Compute query-key affinity scores "comparing" Q and K , e.g., $S = QK^T \in \mathbb{R}^{L \times L}$ (dot-product affinity)
- Convert these scores to probabilities (row-wise): $P = softmax(S) \in \mathbb{R}^{L \times L}$
- Output the weighted average of the values: $Z = PV = softmax(QK^T)V \in \mathbb{R}^{L \times d_v}$
- $q = 1/\text{dimensao} * (Z^T) * \mathbf{1}$
- attention probabilities = softmax($1/\sqrt(d_k)$Image feature matrix $\cdot$ query vector)

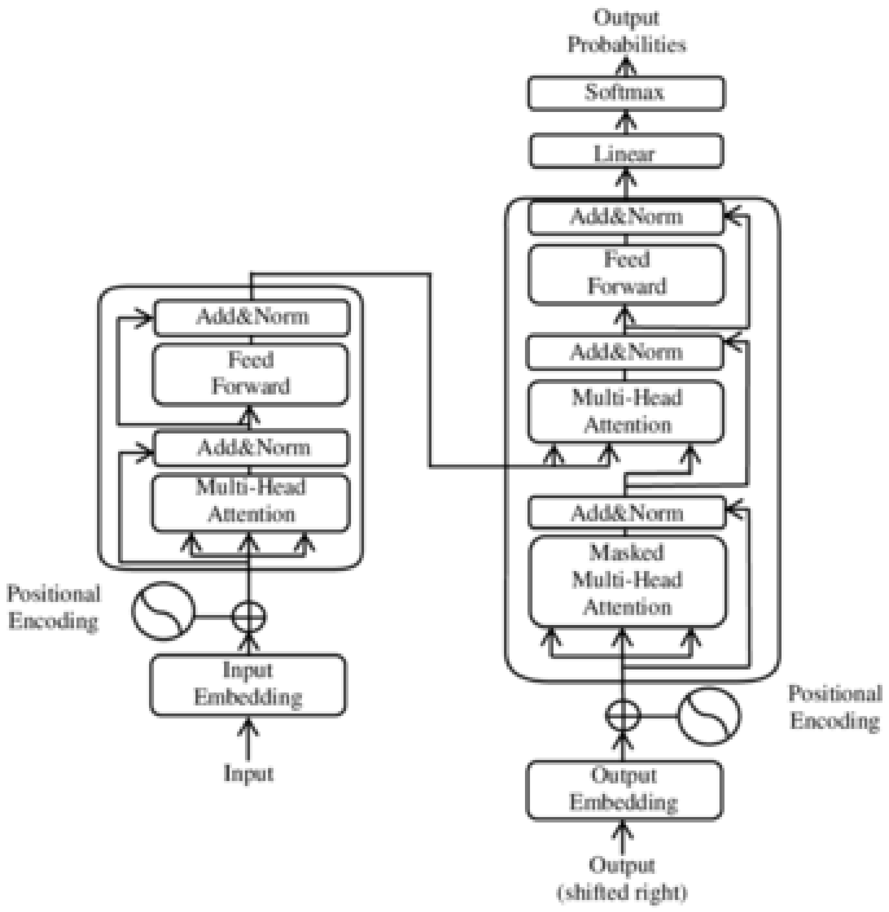


Figure 4: Transformer architecture

6.3 Transformers

Key Idea: instead of RNN/CNNs, use self-attention in the encoder; Each word state attends to all the other words; Each self-attention is followed by a feed-forward transformation; Do several layers of this; Do the same for the decoder, attending only to already generated words
What's new? scaled dot-product attention and multi-head attention

Query, Key and Value vectors obtained by projecting the embedding matrix X $\in \mathbb{R}^{L\times e}$ to a lower dimension: $Q = XW_Q$, $K = XW_K$, $V = XW_V$. The projection matrices W^Q, W^K, W^V are model parameters. Dimensions are all L rows, d columns, where d is the representation depth

6.3.1 Scaled Dot Product

Problem: As d_K gets large, the variance of $q^T k$ increases, the softmax gets very peaked, hence its gradient gets smaller.
Solution: scale by length of query/key vectors: $Z = softmax(\frac{QK^T}{\sqrt{d_k}})V$

6.3.2 Multi-Head Attention

Self-Attention Idea: each word forms a query vector and attends to the other words' key vectors. Somewhat similar to a 1D convolution, with "adaptive" weights and the window size spans the entire sentence

Problem: only one channel for words to interact with one-another
Solution: multi-head attention!

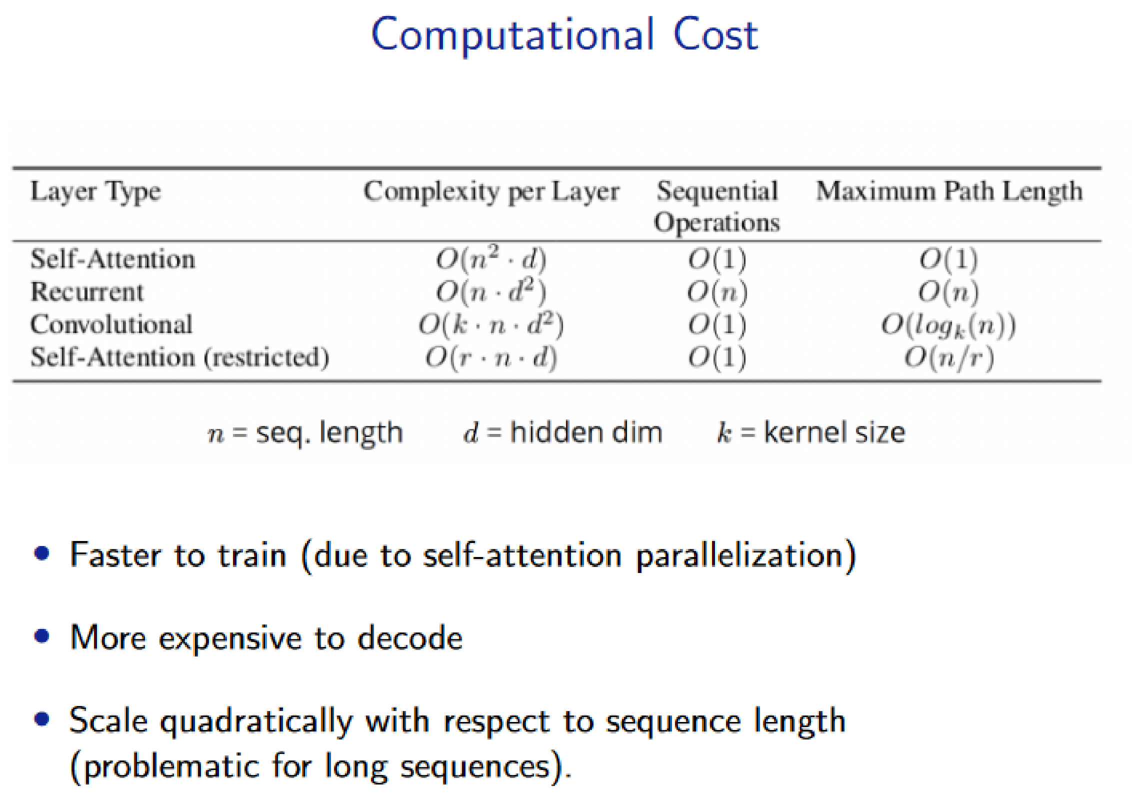
- define h attention heads, each with their own projection matrices
- apply attention in multiple channels, concatenate the outputs and pipe through linear layer: MultiHead(X) = Concat$(Z_1, ..., Z_h)W$ where $Z_i = Attention(XW_i^Q, XW_i^K, XW_i^V)$

Other useful tricks include: repeating self-attention blocks, residual connections on each attention block, layer normalization, positional encodings (to distinguish word positions)

Self Attention in the Decoder? The decoder cannot see the future! Use **"causal" masking** (We give the target input into the transformer decoder while training the model. So it is easy for the model to "peek ahead" and learn what the next word would be. Causal masking prevents this) and the decoder should attend to itself (self-attention), but also to the encoder states (contextual attention).

6.3.3 Positional Encodings

The transformer is insensitive to word order (queries attend to keys regardless of their position in the sequence). To make it sensitive to order, we add positional encodings. Two strategies: learn one embedding for each position (up to a maximum length) or use sinusoidal positional encodings.



Computational Cost

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

n = seq. length d = hidden dim k = kernel size

- Faster to train (due to self-attention parallelization)
- More expensive to decode
- Scale quadratically with respect to sequence length (problematic for long sequences).

7 Self-Supervised Learning and Large Pretrained Models

Contextualized Embeddings: Words can have different meanings, depending on which context they appear in (a model called ELMo learned context-dependent embeddings). **Elmo Key Ideas**: Pre-train a BiLSTM language model on a large dataset; Save all the parameters at all layers, not only the embeddings; Then, for your downstream task, tune a scalar parameter for each layer, and pass the entire sentence through this encoder

7.1 Pre-Training and Fine-Tuning

Pretraining through language modeling: Train a NN to perform language modeling on a large amount of text and save the network parameters. Pretraining can be very effective by serving as **parameter initialization**.

- Step 1: Pretrain to learn general things
- Step 2: Finetune (on your task), with fewer labels, to adapt to the task

Pretraining and fine-tuning from a "training neural nets" perspective:

- Pretraining leads to parameter $\Theta \approx argmin_{\hat{\theta}} L_{pretrain}(\theta)$
- Fine-tuning approximates $argmin_{\theta} L_{finetune}(\theta)$

- Pretraining helps because SGD stays (relatively) close to $\hat{\theta}$ during fine-tuning
- Pretraining on large datasets exposes the model to many words and contexts not seen in the fine-tuning data (a form of weak supervision)

There are 3 architectures for pretraining: Decoders (Language model, ice to generate from; can't condition on future words), Encoders (Bidirectional context), Encoder-Decoder

Limitations of Fine-Tuning: Fine-tuning a very large model to several tasks can be very expensive and requires a copy of the model for each task

Dangers of Large Pretrained Models:

- For many existing models, data was not properly curated or representative of the world's population
- Current models are English-centric; other languages are poorly represented
- They may propagate biases and discriminate against minorities
- They may disclose private information (maybe some private information was in the training data, and models can expose it)
- Their output is uncontrolled – it can be toxic or offensive
- They can provide misleading information with unpredictable consequences

7.1.1 Pretrained Decoders

When using language model pretrained decoders, we can ignore that they were trained to model $p_\theta(x_t|x_{1:(t-1)})$. Fine-tuning by training a classifier on the last hidden state. $h_1, ..., h_L = Decoder(x_1, ..., x_L)$, $y = softmax(Ah_l + b)$, where A and b are randomly initialized and learned by the downstream task

Two common choices for fine-tuning: Freeze the pretrained model and train only A and b Or fine-tune everything, letting gradients backpropagate through the whole network

7.1.2 Pretrained Encoders

Encoders get bidirectional context, so we can't do language modeling, we'll have to pretrain with **Masked Language Modeling**.

Masked Language Modeling Idea: replace a fraction of words in the input with a special [MASK] token; predict these words. Only add loss terms from words that are "masked out." If $\tilde{x}$ is the masked version of x, we're learning $p_\theta(x|\tilde{x})$. Similar to a denoising auto-encoder

Limitations: pretrained encoders don't naturally lead to nice autoregressive generation methods, so we shouldn't use them to generate sequences
Use cases: If your task involves classification or sequence tagging, use a pretrained encoder; you can usually benefit from bidirectionality

7.1.3 Pretrained Encoder-Decoder

For encoder-decoders , we can do something like language modeling, but where a prefix of every input is provided to the encoder and is not predicted $h_1, .., h_L = Encoder(x_1, ..., x_T)$, $h_{T+1}, .., h_{2T} = Decoder(x_{T+1}, ..., x_{2T})$, $y_i = softmax(Ah_i + b)$, i ¿ T. The encoder portion benefits from bidirectional context and the decoder portion is used to train the whole model through language modeling

7.2 Self-Supervised Learning

Pretraining on language model task is a form of self-supervised learning

- Take raw (unlabeled) data, remove information and train a model to recover that information
- In the case of language modeling, the information removed is the next word; the model is trained to predict future words given the context
- Other strategies: mask words

7.3 Adapters

Alternative to fine-tuning language models on a downstream task. Instead of fine-tuning the full model, a small set of task-specific parameters (adapter) is appended to the model and updated during fine-tuning. They achieve high performance in downstream tasks with much fewer new parameters

Advantages: Much fewer parameters to fine-tune; Can share the same big pretrained model across tasks, and fine-tune only the task-specific adapters; Can also be used to create multilingual models

7.4 Few-Shot Learning

Used to solve a completely new task for which not enough data exists, not even for fine-tuning. Models like GPT-3 do this via **prompting**.

Pretrained language models acquire a lot of factual knowledge, this suggests we can prompt them on-the-fly to solve new tasks

8 Misc

One-hot vector: ϕ(x) ∈ 1, 2,..., K -¿ ϕ(x) = [0, ..., 0, 1, 0, ..., 0] ∈ $(0,1)^K$

Overfitting occurs when the *training accuracy increases, while the test accuracy decreases*. We can avoid overfitting by increasing regularization, increasing the number of labeled examples in the training set or reducing complexity (number of parameters). l_2 (ridge) regularization promotes smaller weights and l_1 (lasso) regularization promotes smaller and sparse weights. l_2 regularization is equivalent to a Gaussian prior on the weights (weights follow a Gaussian), whereas l_1 is equivalent to a Laplacian prior.

Early Stopping: To select the number of epochs, stop training when validation error increase. One common strategy (with SGD) is to halve the learning rate for every epoch where the validation error increases

Data Normalization: For each input dimension: subtract the training set mean and divide by the training set standard deviation. It makes each input dimension have zero mean, unit variance. Doesn't work for sparse inputs (destroys sparsity)

NNs Hierarchiquical Compositionality: deep(er) NNs learn coarse-to-fine representation layers (pixels -¿ edges -¿ motifs -¿ parts -¿ objects); Layers closer to inputs learn low-level representations, layers farther away from inputs learn more abstract representations

Distributed Representation: one dimension per property, no single neuron "encodes" everything; groups of neurons (e.g. in the same hidden layer) work together. They are more compact and also more powerful, as they can generalize to unseen objects in a meaningful way. Hidden units should capture diverse properties of objects (not all capturing the same property), this is usually achieved by random initialization of the weights

tanh(z) = $\frac{e^z - e^{-z}}{e^z + e^{-z}}$

8.1 Auto-Encoder

Auto-encoder: feed-forward NN trained to reproduce its input at the output. Can also be useful for: Dimensionality reduction, Information retrieval and semantic hashing and Conversion of discrete inputs to low-dimensional continuous space
Key idea: learn the manifold where the input objects live
Encoder: $h(x) = g(Wx + b)$
Decoder: $\hat{x} = W^T h(x) + c$
Loss function (for x $in\mathbb{R}^D$): $L(\hat{x}, x) = ||\hat{x} - x||^2$
Latent Representation: is the lowest one, smallest hidden layer **Output layer**: is N^2 where input is NxN

8.1.1 Auto-Encoder Variants

Regularized Auto-Encoder: regularization may be added to the loss. The goal is then to minimize $L(\hat{x}, x) + \Omega(h, x)$, for example $\Omega(h, x) = \lambda ||h||^2$ or $\Omega(h, x) = \lambda \sum_i ||\nabla_x h_i||^2$
Sparse Auto-Encoders: use many hidden units, but add a l_1 regularization term to encourage sparse representations of the input, where most hidden units are inactive.
Denoising Auto-Encoders: regularize by adding noise to the input; the goal is to learn a smooth representation function that allows to output the denoised input (inspired by image denoising). Basically, apply noise to the input $\tilde{x} = x + n$ and minimize

$\frac{1}{2}||\hat{x} - \tilde{x}||$ instead of $\frac{1}{2}||\hat{x} - x||$. Encourages to represent well the data points and their perturbations.
Stacked Auto-Encoders: several auto-encoders on top of each other
Variational Auto-Encoders: a generative probabilistic model that minimizes a variational bound

8.1.2 Variational Auto-Encoders

VAEs consist of an encoder network that maps input data to a probability distribution in the latent space, and a decoder network that reconstructs the input data from samples drawn from this distribution. VAEs introduce a probabilistic interpretation to traditional autoencoders by treating the latent variables as random and imposing a specific prior distribution. The training process involves optimizing a trade-off between accurately reconstructing input data and regularizing the learned latent space to follow the desired prior distribution, typically a multivariate Gaussian. This regularization encourages the model to learn a continuous and smooth latent representation

8.2 Chain Rule

Take r = uv, u = t^2, v = 3t + 1, then we have $\frac{\partial r(t)}{\partial t} = \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t}$

8.3 Common Derivatives

sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
hyperbolic tangent: $tanh'(z) = 1 - tanh^2(z)$
Rectified Linear Unit: relu'(z) = $1_{z>0}$
Natural Logarithm: ln'(f(x)) = $\frac{1}{f(x)}\frac{\partial f(x)}{\partial x}$

8.4 Initialization Tricks

Bias: Initialize to 0
Weights: **Cannot initialize to zero with tanh activation** (gradients would also be all zero and we would be at saddle point); **Cannot initialize the weights to the same value** (need to break the symmetry); **Random initialization (Gaussian, uniform), sampling around 0** to break symmetry; For ReLU activations, the mean should be a small positive number; **Variance cannot be too high**, otherwise all neuron activations will be saturated
For Batch Gradient Descent or Stochastic Gradient Descent:
If all weights are initialized to zero, all neurons in all layers will receive the same gradient and thus update their weights in the same way. This phenomenon is often referred to as the "symmetry problem", essentially implying that all neurons in a layer will learn the same thing, negating the benefits of having multiple neurons in a layer.

8.4.1 Hyperparameter Tuning

Grid Search: specify a set of values to test for each hyperparameter, and try all configurations of these values
Random Search: specify a distribution over the values of each hyper-parameter (e.g. uniform in some range) and sample independently each hyper-parameter to get configurations
Bayesian Optimization

8.5 Skip/Residual Connections

A Residual Neural Network is a deep learning model in which the weight layers learn residual functions with reference to the layer inputs. A Residual Network is a network with skip connections that perform identity mappings, merged with the layer outputs by addition. This enables deep learning models with tens or hundreds of layers to train easily and approach better accuracy when going deeper.

They are used to allow gradients to flow through a network directly, without passing through non-linear activation functions. Non-linear activation functions, by nature of being non-linear, cause the gradients to explode or vanish (depending on the weights).

8.6 Squared Loss

As in 2, squared loss is often defined as $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$. The constant $\frac{1}{2}$ doesn't really matter and is used just to help with some computations. The squared loss itself is $L(y, \hat{y}) = (y - \hat{y})^2$. Multiplying by a constant doesn't really affect what we are trying to do, because we are usually trying to find the parameters that minimize the loss $argmin_w L(y, \hat{y}) = (y - \hat{y})^2$. Scaling the loss function won't change the optimal parameterization, but it will help when taking the loss function's gradient, as the $\frac{1}{2}$ will cancel out with the square of the loss:

$$\nabla_W L(y, \hat{y}) = \nabla_W \frac{1}{2}(y - \hat{y})^2 =$$
$$= \frac{1}{2} \times 2(y - \hat{y})\nabla_W (y - \hat{y}) =$$
$$= (y - \hat{y})\nabla_W (y - W^T x) =$$
$$= (y - \hat{y})(\nabla_W y - \nabla_W W^T x) =$$
$$= (y - \hat{y})(-x) =$$
$$= x(\hat{y} - y)$$

8.7 Cross-Entropy Derivative

The sum was omitted, because it doesn't change the process of taking the derivative (the derivative is a linear transformation). A sigmoid activation function is assumed here.

$$\nabla_W y log(\sigma(W^T x)) + \nabla_W (1 - y) log(1 - \sigma(W^T x) =$$
$$= y\frac{\nabla_W \sigma(w^T x)}{\sigma(w^T x)} + (1 - y)\frac{\nabla_W (1 - \sigma(W^T x))}{1 - \sigma(W^T x)} =$$
$$= y\frac{\sigma(W^T x)(1 - \sigma(W^T x))x}{\sigma(W^T x)} + (1 - y)\frac{-\nabla_W \sigma(W^T x)}{1 - \sigma(W^T x)} =$$
$$= y\frac{\sigma(W^T x)(1 - \sigma(W^T x))x}{\sigma(W^T x)} - (1 - y)\frac{\sigma(W^T x)(1 - \sigma(W^T x))x}{1 - \sigma(W^T x)} =$$
$$= y(1 - \sigma(W^T x)x - (1 - y)\sigma(W^T x)x =$$
$$= x(y - y\sigma(W^T x) - \sigma(W^T x) + y\sigma(W^T x)) =$$
$$= x(y - \sigma(W^T x))$$

TODO - inductive bias

8.8 Additional Notes from Past Exams:

- **Final Layer for Classification Task**: fully connected linear layer followed by softmax activation function.
- Within the same batch, all sequences must have the same length, and for this reason they must be padded with padding symbols for making them as long as the longest sentence in the batch. Since in NLP sentences can have very different lengths, if we don't sort sentences by length, we can end up with very unbalanced batches, where some sentences are very short and others are very long, which makes it necessary to add a lot of padding symbols. This process is inefficient and can make training more time consuming.
- Gradient descent (stochastic or batch) may converge or not, depending on the choice of step size, and its convergence, even with an adequately chosen step size, has no guarantees of providing a separating hyperplane in a finite number of iterations.
- By applying the same filter across the entire input, a convolutional layer detects the same features regardless of position, making them translation equivariant. Also, they assume that small, localized regions of the input are relevant for feature detection, leading to the use of local receptive fields.
- GPT-3 is a decoder-only model. BERT is an encoder-only model trained on a masked language modeling objective.
- XOR = (A¬B)(¬AB) cannot be learned by single perceptron, cannot be separated in hyperspace plane
- Neural networks with a single hidden layer with linear activations are equivalent to linear classifiers, which are not universal approximators.
- Auto-encoders with linear activations would correspond to PCA.

- CNNs take advantage of the spacial structure of the image, unlike standard feed- forward networks. Moreover, convolutional and pooling layers exploit the fact that the same feature may appear in different parts of the image, enabling the network to process those occurrences in a similarly way.
- Bart could use a pretrained decoder-only (e.g., GPT) or encoder-decoder model (e.g., T5, BART) and fine-tune it on the data he has available. Alternatively he could use the model without any fine-tuning and use prompting at test time. A possible prompt would be "¡document¿ TL;DR: ¡answer¿". Note: an encoder-only model (e.g. BERT) would not be suitable, since this an auto-regressive generation task.
- Linear regression: This is used for continuous target variables, not for classification tasks. Binary logistic regression: This is suitable only for classification tasks with two labels (binary classification). Multi-class logistic regression: This is specifically designed for classification tasks with more than two labels.
- Decoder-only models, including GPT, are trained with a causal language modeling objective.
- Prompting is a more efficient way to adapt models than fine- tuning, but it is usually done with models such as GPT, which are trained with a causal language modeling objective.
- An adapter is a lightweight module added to a pre-trained model for task-specific learning, freezing the main model and training only the adapter layers. Advantages over fine-tuning: Adapters are parameter-efficient, prevent catastrophic forgetting, and allow faster, scalable multi-task learning by reusing the same base model.